THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 61, No. 9, November 1982
Printed in U.S.A.

Database Systems:

The Intelligent Store: A Content-Addressable Page Manager

By W. D. ROOME

(Manuscript received September 29, 1981)

The Intelligent Store (IS) is a page manager for an experimental local computer network. Typical clients of the IS are a database manager or a file manager. The IS provides concurrent, transaction-oriented, read-write-search access to a large database of pages. The IS ensures that committing a transaction is atomic and permanent, in spite of crashes. The IS allows multiple, concurrent readers and writers. Rather than using conventional wait-until-available locking, the IS uses an optimistic locking policy. That is, the IS always allows transactions to proceed, assuming they will not interfere. If two transactions do interfere (e.g., if they both write the same page), the IS aborts one of them. This paper gives some analytic results for the interference probability. Two implementations of the IS exist. The first was designed to use one or more content-addressable disks. By reading all tracks in a cylinder in parallel, and filtering each stream, such a device can search an entire cylinder in one disk revolution. This version of the IS uses the content-addressable features internally, as well as making them available to its clients. Unfortunately, suitable content-addressable disks are not commercially available. Therefore, a second implementation of the IS was designed to use conventional disks. The two versions of the IS present the same interface to their clients, except that the conventional-disk IS does not provide search access.

I. INTRODUCTION

1.1 Overview

The Intelligent Store (IS) is a page manager for an experimental

local computer network.[1] Typical clients of the IS are a database manager or a file manager. The IS provides concurrent, transaction-oriented, read-write-search access to a large database of pages. If a hardware or software crash occurs, the IS restores the database to its most recent consistent state.

Note that the IS is a low-level component of a Database Management System (DBMS). Normally there will be several additional layers of DBMS software between the IS and end users or application programs.

The IS is transaction-oriented. Clients can start transactions, read or write pages for transactions, or commit or abort transactions. Many transactions can be active simultaneously, and each transaction can read and write an arbitrary number of pages. Writes for a transaction are not effective until (and unless) the transaction commits. The IS ensures that commit is atomic and permanent, in spite of crashes. "Atomic" means that either all writes become effective, or else none do. "Permanent" means that once the IS says that a transaction has been committed, its writes will not be lost, even if the IS crashes immediately.

Aborting a transaction undoes all its effects. If a client crashes, the IS automatically aborts all of that client's uncommitted transactions. If the IS crashes, the IS will (eventually) abort all uncommitted transactions. Furthermore, the IS reserves the right to abort any uncommitted transaction at any time (for cause, of course, but the client has no appeal).

The IS ensures that transactions do not interfere with each other. Rather than using conventional wait-until-available locking,[2,3] the IS uses an optimistic locking policy, similar to that of Kung and Robinson.[4] That is, the IS always allows transactions to proceed, assuming they will not interfere. If two transactions do interfere (e.g., if they both write the same page), the IS aborts one of them. Subsequent sections give analytic results for the probability that the IS will abort a transaction.

A page contains one or more variable-size records. Pages have a fixed maximum size (say 512 to 4096 bytes) set by a system administrator. The IS assigns a unique page number to each page, and clients can read or write pages by number. These page numbers are not physical disk block addresses.

There are two implementations of the IS. The first was designed to utilize one or more parallel-search disks, and to demonstrate that they could be used effectively in an update environment. A parallel-search disk can search all tracks in a cylinder in one disk revolution. It is functionally similar to the Mass Memory described by Banerjee, Hsiao, and Kannan,[5–7] or to the search engine in the Content-Addressable File Store[8] by International Computers Limited. This version of the IS

used the content-addressable features internally, in addition to making them available to the client. For example, the client could ask the IS to retrieve all pages that contain a record with NAME = "Smith" and DEPTNO = 1234.

Unfortunately, parallel-search disks are not yet commercially available (at least not at a reasonable price). We considered building our own prototype of a parallel-search disk. However, it would have been a year before we could use it, and at least another year or two before anyone else could use it.

Because we wanted to demonstrate that the IS could handle large databases efficiently, we temporarily shelved the plans for a prototype parallel-search disk. Instead we designed a second implementation of the IS, this time based on conventional disks. The two versions of the IS have the same interface to the client, except that the conventional-disk version does not provide search access. When it matters, I'll refer to these as the "content-addressable" and "conventional-disk" versions of the IS.

The network may have several distinct IS's. An IS can partition its page database so that several different clients, each with a small database, can share the same IS. Conversely, a client with a very large database could use several IS's. In this case, the IS's cooperate to provide the same consistency and atomic-commit properties as for a single IS. The IS's do not hide the distribution from clients: clients must know which IS holds which page. Clients may keep multiple copies of data, but synchronizing those copies is the clients' responsibility.

1.2 Network environment

The IS is one component in an experimental local network of small computers.[1] A network kernel provides intertask communication services, and does basic task scheduling. The kernel allows families of tasks to run on a processor. All tasks in a family share a common address space. Several families may share a processor, but all tasks in a family must run on the same processor. For example, the IS is implemented as a family of about 15 tasks. The clients of the IS would be in one or more separate families.

Ideally, each IS would run on a dedicated processor, as would each client. Figure 1 shows a typical configuration. Here the IS's client is a Database Management System (DBMS). However, the IS does not require a dedicated processor. In the initial prototype, the IS, the DBMS, and its clients all ran on the same processor.

The kernel provides a uniform, path-based, message-passing communications mechanism, similar to that used in the DEMOS[9] and Roscoe[10] systems. The kernel uses whatever physical connections are

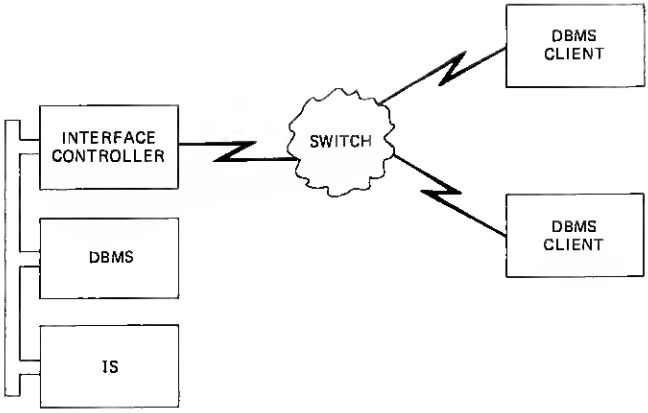

Fig. 1—Typical network configuration.

available. For example, in Fig. 1, a packet switch connects the DBMS to its clients, but the IS and DBMS processors are connected by a high-speed bus. The kernel hides this nonuniformity, and provides tasks with a uniform communications mechanism. Thus, the IS uses the same mechanism to talk to the DBMS as the DBMS uses to talk to its clients. Replacing the high-speed bus might degrade performance, but wouldn't require changes to the IS or DBMS.

The kernel provides simple, nonpreemptive task scheduling: a task runs until it needs to receive a message. The kernel does not provide swapping, time-slicing, or I/O. The kernel turns interrupts from devices into messages to tasks.

1.3 Organization

Section III describes the consistency model, and Section IV describes the interface which the IS presents to its clients. Together, these define the abstract "IS machine," without describing how it's implemented. Section V describes how to implement the IS with parallel-search disks, and Section VI describes how to implement the IS with conventional disks. But first we need some philosophy.

II. DESIGN CHOICES

This section discusses several fundamental design decisions. These decisions have influenced the services which the IS provides, as well as the way those services are implemented.

2.1 Workload

The IS is intended for applications where most transactions are small (read and write a total of 2 to 15 pages), and fast (after reading

a page, a client doesn't wait 30 minutes before writing it). The IS can handle long-running read-only transactions, provided that they are run during periods of low update activity.

However, the IS is not suitable for all environments. In particular,

- If there is a lot of read/write interference between transactions, the IS will perform poorly. In this case, the client can use an external lock manager to control such interference (see Section 3.2).
- The IS has a fixed limit on the total number of pages that can be written during the lifetime of the longest-running transaction. This limit is typically 10 to 20 percent of the size of the database. Thus, the IS cannot handle single transactions which update a large fraction of the database. Furthermore, the IS cannot handle long-running transactions if, during their lifetime, other transactions update a large fraction of the database.

2.2 *Why a content-addressable device?*

Several research projects on database machines have built or proposed content-addressable memory devices.[11–16] A typical device is a fixed-head disk with n tracks, with a search filter on the output of each track. Equivalently, a set of n CCD (charge coupled device) arrays or magnetic bubble loops can be used. The n filters search in parallel, and a multiplexor merges their output streams. Thus, the device can search the entire disk in one revolution. Complexity of the search expressions varies, but most allow at least the OR of AND's of several simple predicates of the form:

⟨field-name⟩⟨operator⟩⟨constant-value⟩

Then given a simple query such as:

NAME = "Smith" & DEPTNO = "1234"

the database machine loads the query into the filters, and returns the output stream to the client. One advantage of such systems is that indexes are unnecessary.

Unfortunately, those systems do not scale up nicely to large databases.[17–19] For large databases, moving-head disks are still the most cost-effective devices, and are expected to remain so for many years. The obvious extension is to put a filter on each head, so that a cylinder can be searched in one revolution. Typically a cylinder is 10 to 20 tracks (100K bytes to 400K bytes). Then to search the database, the database machine loads the query into the filters, and searches each cylinder. However, that's too slow for a large database. It takes about 24 ms to search each cylinder (17 ms per revolution, 7 ms to step to the next cylinder). For a database of 75 cylinders, each query will take 1.8 seconds. Since the disk is totally dedicated to the query, such a

system can handle at most 33 queries per minute. This isn't acceptable for many applications. A simple indexing scheme on a conventional disk can do much better.

The solution is to use a cylinder-level index.[8,19] Consider the NAME and DEPTNO query above. For example, the NAME index might tell us that records with NAME "Smith" appear in cylinders 4, 13, and 42. The DEPTNO index might say that records for department "1234" appear in cylinders 3, 6, 13, 27, and 31. Cylinder 13 is the only one on both lists, so it is the only cylinder we need to search.

Cylinder-level indexes have several advantages over conventional record-level indexes. In particular, they take less space, and they needn't be updated as often. For example, consider a value such as "Smith" which appears in many records. A record-level index would need the address of each such record. A cylinder-level index only needs the distinct cylinders that contain those records. As another example, suppose we add a new "Smith" record. With a record index, we would always have to add an index entry for the new record. But a cylinder index doesn't need to be updated, as long as we add the new record to a cylinder that already has a "Smith."

One further advantage is that a parallel-search disk does allow you to search the entire database occasionally. For example, consider a database with NAME and STREET-ADDRESS fields. Queries on NAME are very common, so NAME must be indexed. But queries on STREET-ADDRESS are rare, perhaps one per day. Rather than indexing the STREET-ADDRESS field, it might be better to search everything for the occasional STREET-ADDRESS query. That query might take a few minutes, but the total overhead for maintaining another index would be at least a few minutes per day. The DBMS should be flexible enough that a STREET-ADDRESS index can be added later, if such queries become common.

2.3 Why pages?

Most clients don't care about pages. They really want stable storage for variable-size objects. Such objects include:

(*i*) Small to medium size data structures (4 to 400 bytes each);
(*ii*) Records in a database (50 to 500 bytes);
(*iii*) Files (10 to 100,000 bytes).

So why isn't the IS an object manager, instead of a page manager? The problem is that large storage devices (e.g., disks) are block-oriented. That is, storage is really a collection of blocks, on the order of 1000 bytes each. Assessing 10 bytes in a block is no faster than accessing the entire block. Something must pack small objects into these blocks, and vice versa for large objects. For fast retrieval, related objects should be packed into the same block. To save space, objects

should be packed efficiently. Such packing is highly dependent on the nature of the objects and their reference patterns, and has been left to the client.

2.4 Disk failure modes

Disks are not perfect storage devices: blocks can get corrupted, destroying the data in them. For example, updating a block is inherently dangerous. Since writing is a bit-serial process, if any error occurs while writing (system crash, power flicker, glitch in the computer's memory, etc.), both the old and the new version of the block may be lost. Let's call this synchronous corruption. The other basic failure mode is asynchronous corruption: a block or set of blocks spontaneously gets corrupted. A head crash is the classic example.

There are many well-known ways to recover from these kinds of failures.[3,20,25] All are based on duplication, in one way or another, and all involve trade-offs between cost, overhead during normal operation, and speed of recovery from a failure. For example, intention-lists can be used to recover from synchronous failures. They duplicate only what is being changed. Another solution is to duplicate each disk, and write to both. This is expensive, but it allows fast recovery from both synchronous and asynchronous failures. A cheaper method is to periodically copy all disks to some other media, such as tape. If a failure occurs, the disks would be restored from the latest copy. Changes since the copy can be kept in a log file, and reapplied. An even cheaper solution is to ignore the problem altogether (at least it seems cheaper at first).

The following assumptions have influenced the design of the IS:

(*i*) Synchronous failures are much more likely than asynchronous failures (at least when writing large blocks).

(*ii*) No client wants to ignore synchronous failures.

(*iii*) All clients want fast recovery from synchronous failures.

Thus, the IS provides only one mechanism for recovering from synchronous failures: basically a super-shadowing technique. This mechanism is a central design choice because it allows the consistency model described in Section III to be implemented with very little extra overhead.

A further assumption is that asynchronous failures are sufficiently rare, and the class of potential clients is sufficiently broad, that most clients do not want fully duplicated disks. To protect against asynchronous failures, the IS uses periodic copies, and an optional log file.

III. THE CONSISTENCY MODEL

3.1 Definition

The IS guarantees each transaction a consistent image of the entire

database of pages, as of when the transaction started. That is, if transaction T starts at time t, T sees the results of all transactions that commit before time t, but does not see the results of any transactions which commit after time t. Thus T sees a consistent, but possibly out-of-date image. Of course, T sees its own updates. The effect is as if, when T starts, the IS made a separate copy of the database for transaction T. However, that is not how it is implemented (see Section V).

The IS does not have conventional read or write locks. The IS allows any transaction to read or write any page at any time, without waiting. Instead of conventional locks, each transaction declares certain pages to be important. Normally these are all the pages that the transaction reads or writes. To be precise, the important pages are those that the transaction would lock for reading or writing in a conventional system.[2,3]

When the client asks the IS to commit transaction T, the IS first verifies that, for each important page, the version given to transaction T is still current. If so, the IS commits T. If not, the IS aborts T, and tells the client "tough luck: try again." More formally, the IS allows transaction T to commit if and only if, for all transactions T_i which committed during the lifetime of T,

$$I \cap W_i = \phi,$$

where I is the important-set for T, and W_i is the write-set for T_i. In effect, the IS tries to serialize transactions in the order in which they commit. The IS aborts any transaction that does not fit this schedule.

Figure 2 gives an example. Transaction A creates page P, and B updates it. Transactions C, D, and E also update page P. Because C starts before B commits, C sees the version of P written by transaction A. D and E start after B commits, so they see what B wrote. If C considers page P to be important, C cannot commit. If D and E both consider P important, one of them will commit, and the other will abort.

As another example, consider a long transaction that reads all account balances at a bank, and writes their sum: "at 3PM, the total balance was ...". The page (or pages) into which the transaction writes the total is important. But the other pages that the transaction reads aren't important. Since the IS guarantees a consistent image, the total balance will be correct, even if some accounts are updated while the transaction is active.

Subsequent sections give a rationale for this model, an analysis of the interference probability, and some alternatives. Note that if you replace "page" by "object" (or "record"), this consistency model, and the analysis, apply just as well to a general object manager.

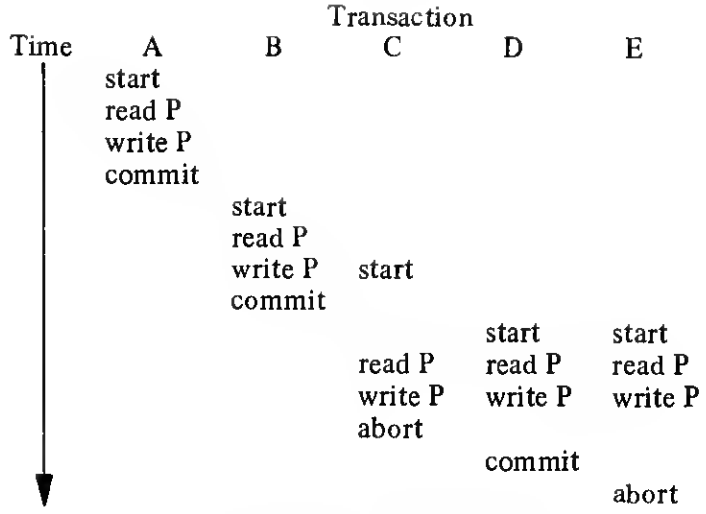

Fig. 2—An example of the consistency model.

3.2 Why this model?

This model has two advantages:

(*i*) It can't deadlock.

(*ii*) A long-running transaction can't hold up other transactions. The latter means that the IS can afford long time-out limits on transactions. With a conventional locking scheme, a transaction can delay others by holding a lock for a long time. With the IS, such a transaction can hurt itself, because it's less likely to succeed when it finally commits, but it can't hurt any other transaction.

Of course, this model assumes that there is little interference between transactions. We feel that this is natural for many database applications (see the analysis in the next section). Nevertheless, there are applications with high interference, and they will need conventional wait-until-available locking. The IS could provide such locking. However, a better solution is an external Lock Manager, outside the IS, and logically above it. Before going to the IS, a transaction would get outside locks to cover its important pages. An external lock manager has the following advantages:

(*i*) The optimal locking granularity depends on the application.[21,22] The IS would have to lock pages; an external lock manager can be tailored to the application.

(*ii*) The external lock manager can be *approximate*, rather than exact. The lock manager doesn't control consistency; it just decreases interference between transactions as seen by the IS. Thus a transaction only needs to acquire outside locks to cover most of its pages.

3.3 Analysis

This section investigates the probability that a given transaction T will not be able to commit. This probability is given as a function of various parameters of T, such as the number of important pages and the running time. First, here are a few assumptions and definitions:

(*i*) Let I be the set of important pages for transaction T.

(*ii*) Let t be the lifetime of T, in seconds (start-request to commit-request).

(*iii*) Assume that update transactions commit and depart from the IS according to a Poisson distribution, averaging λ transactions per second. The quantity λ does not include read-only transactions.

(*iv*) Assume that I is independent of the write-sets of the transactions that commit during the life of T, and assume that those write-sets are independent of each other.

(*v*) Let p be the probability that any one transaction interferes with T:

$$p \equiv \text{Prob}(I \cap W \neq \phi),$$

where W is the write-set of an arbitrarily selected update transaction.

In general, the parameters p and t both increase as I increases. The parameter p also depends on the average number of pages written by transactions, and on the distribution of page references. For example, hot-spots in the database might increase p.

The average number of transactions that commit during T's lifetime is λt. Then the probability that exactly k transactions commit is:

$$\text{Prob}(C_k) \equiv \frac{(\lambda t)^k}{k!} e^{-\lambda t}.$$

Because the committing transactions are independent of I, if exactly k transactions commit, the probability that none of those k transactions interfere with T is:

$$\text{Prob}(X_k) \equiv (1-p)^k.$$

Now for the probability that T will fail:

$$\begin{aligned}
\text{Prob}(T \text{ aborts}) &= 1 - \text{Prob}(T \text{ commits}) \\
&= 1 - \sum_{k=0}^{\infty} \text{Prob}(C_k)\text{Prob}(X_k) \\
&= 1 - \sum_{k=0}^{\infty} \frac{(\lambda t)^k}{k!} e^{-\lambda t}(1-p)^k \\
&= 1 - e^{-\lambda t} \sum_{k=0}^{\infty} \frac{[\lambda t(1-p)]^k}{k!} \\
&= 1 - e^{-\lambda t} e^{\lambda t(1-p)},
\end{aligned}$$

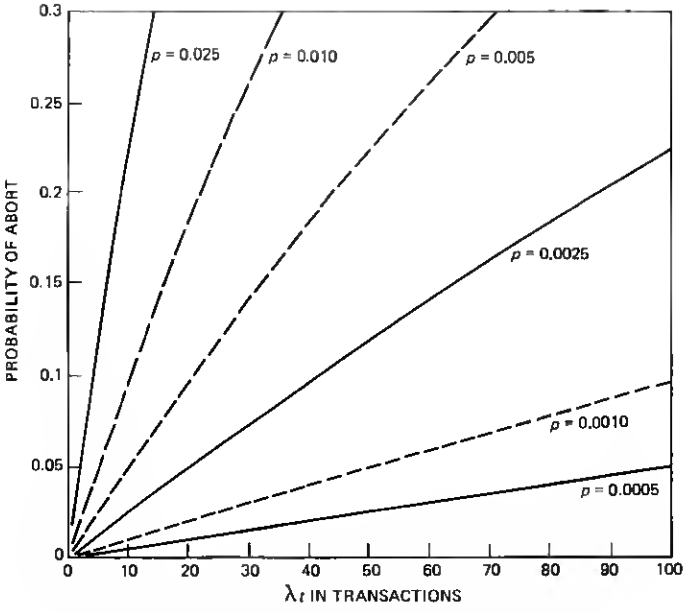

Fig. 3—Probability of abort vs. λt for various p.

which finally gives

$$\text{Prob}(T \text{ aborts}) = 1 - e^{-\lambda t p}.$$

Figure 3 graphs this probability versus λt, for various values of p. Recall that λt is the average number of transactions that commit during T's lifetime. Let's say that a failure probability under 0.05 is acceptable. Then if $p = 0.01$, a transaction should be short enough that no more than five other transactions commit during its lifetime. If $p = 0.001$, a transaction can safely allow 50 other transactions to commit during its lifetime.

Unfortunately, the parameter p is not very intuitive. For the special case of uniform references, p can be expressed in terms of more meaningful parameters. Let's assume that:

(*i*) A write-set W contains m distinct pages that have been uniformly and independently selected from a universe of N pages.

(*ii*) An important-set I contains n distinct pages, selected independently of each other and of the pages in W. However, the pages in I themselves do not have to be selected uniformly.

Given these assumptions, the appendix proves that:

$$p \equiv \text{Prob}(I \cap W \neq \phi) = 1 - \prod_{i=0}^{n-1} \frac{N - (m + i)}{N - i}.$$

Table I gives p for various values of m and n.

Table I—p vs. m and n for selected database sizes

		Database Size (Pages)				
m	n	10,000	50,000	100,000	500,000	1,000,000
5	5	.002500	.000500	.000250	.000050	.000025
5	10	.004991	.001000	.000500	.000100	.000050
5	20	.009962	.001998	.001000	.000200	.000100
5	50	.024756	.004990	.002498	.000500	.000250
5	100	.049020	.009960	.004990	.001000	.000500
5	200	.096098	.019841	.009960	.001998	.001000
10	5	.004991	.001000	.000500	.000100	.000050
10	10	.009960	.001998	.001000	.000200	.000100
10	20	.019830	.003993	.001998	.000400	.000200
10	50	.048911	.009956	.005000	.001000	.000500
10	100	.095659	.019822	.009956	.001998	.001000
10	200	.183002	.039291	.019821	.003992	.001998

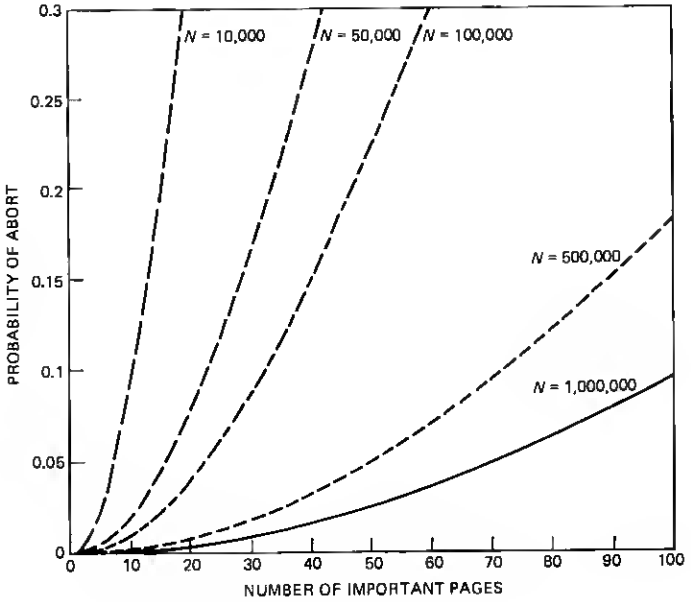

Fig. 4—Probability of abort vs. size of important-set.

Figure 4 graphs the probability of a transaction failing versus the number of important pages, n. Curves are given for several different database sizes, ranging from 10,000 pages to 1,000,000 pages. For all databases, the size of the write-set, m, was assumed to be uniformly distributed between 1 and 9 pages, and λ was assumed to be five transactions per second, for a page-update rate of 25 pages per second. The remaining parameter is T's lifetime, t, which was arbitrarily set to $0.4n$ second.

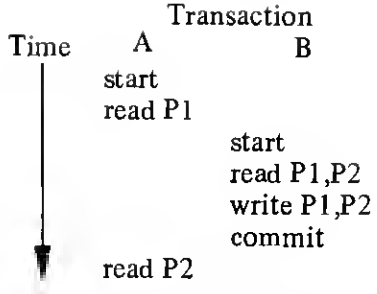

Fig. 5—Transaction A with inconsistent image.

As can be seen, for a database with 1,000,000 pages, only very large transactions (over 70 important pages) run a risk. For a 100,000-page database, a transaction with more than 25 important pages is risky. For a small database with 10,000 pages, only small transactions (less than 8 important pages) are safe. Of course, this page-update rate (1500 pages/minute) is very high for a 10,000 page database.

3.4 Alternative versions of the model

3.4.1 The Kung and Robinson model

When a transaction reads a page, the IS returns the most recently committed version, instead of the version committed when the transaction started. Each transaction declares important pages, as before. When a transaction attempts to commit, the IS verifies that for each important page, the version read by the transaction is still the current version. This is essentially the model described by Kung and Robinson.[4] Because transactions get more recent versions, interference is less likely. This model can be implemented with only a slight change to the IS.

However, this model does not guarantee that a transaction sees a consistent image. For example, suppose that transaction *A* reads page *P1*, pauses, and then reads page *P2* (see Fig. 5). During the pause, transaction *B* starts, updates *P1* and *P2*, and commits. Then *A* sees the old version of *P1* and the new version of *P2*. Fortunately, this does not affect the consistency of the database. If transaction *T* gets an inconsistent image, another transaction must have interfered with it, and hence *T* will be aborted. That is, an inconsistent image implies interference, but not vice versa.

Thus, Kung and Robinson's model precludes transactions that need a consistent image, even if it's not current. One example is the balance-summing transaction described in Section 3.1. With the original IS model, this transaction has only one important page: the one in which it writes the sum. The transaction will almost always succeed. But

with the Kung and Robinson model, the pages for all accounts would be important. The transaction would rarely succeed.

We are considering supporting both models in the IS. When starting a transaction, the client would specify the model for that transaction. That is, the client would say whether it needs (*i*) an absolutely consistent image, or (*ii*) a more current, but possibly inconsistent image.

3.4.2 Early warning

Another suggestion is to tell the client when a transaction becomes a "lame-duck." For example, suppose transaction T reads page P, which has been updated by another transaction that committed after T started. The IS could respond with the page plus a warning such as "by the way, you can't possibly commit." However, there are several problems with this scheme:

(*i*) In general, detecting conflict takes time.

(*ii*) Handling an early abort at an arbitrary place can complicate a client program.

(*iii*) This attacks the wrong problem.

The last point is the most significant. An early warning only helps if the failure probability is already high. A better solution is to decrease that probability, perhaps by using an application-specific lock manager.

3.4.3 Reread dirtied pages

When a transaction can't commit, the IS could tell the transaction which pages had been dirtied by other transactions. The transaction could then reread those pages, make whatever changes are necessary, and try again (this requires the transaction to use Kung and Robinson's model). The Distributed File System described by Sturgis et al allows this.[23] They give an example of how a client can be informed automatically of changes made by other clients.

This feature is easy to implement. However, I can't think of any other practical example for which it's useful. For a simple transaction that reads only a few pages, there's not much difference between rereading a page and restarting the transaction. For a long and complex transaction, rereading would save work. But for such transactions, it would be very difficult to determine how to correct for changes in arbitrary pages. The ability to do that would violate most principles of good program design: information hiding, least privilege, etc.

Again, this feature attacks the wrong problem. Instead, it is preferable to decrease the interference between transactions (draining the swamp solves the alligator problem).

4. INTERFACE TO THE CLIENT

4.1 Client tasks, transactions, and requests

The IS provides service to any number of client tasks. A request is a command that a client task gives to the IS. Examples are start-transaction and read-page. For each request, the client task sends one message to the IS. This message describes the request, and identifies a reply path back to the client task. When the IS has completed that request, the IS sends one or more response messages down that path.

A transaction is a series of requests, beginning with start-transaction, and ending with commit or abort. Each request is on behalf of some active transaction; the message specifies the transaction.

The IS does not have the notion of a controlling task (or a permanent reply path) for a transaction. Any number of cooperating client tasks may work on the same transaction, and they can issue requests concurrently. Requests for the same transaction may have different reply paths. Furthermore, a client task may have several transactions active concurrently.

4.2 Pages and logical volumes

The pages are partitioned into a set of "Logical Volumes" (LV's). Each page is in one (and only one) LV. The IS offers a small number of LV's (perhaps 10 to 50). When the IS is configured, the system administrator specifies various parameters for each LV: page size, maximum number of pages, number of disk blocks reserved for this LV, etc. These parameters may differ between LV's.

This grouping into LV's is really for the client's convenience. For example, it may be useful to have indexes and data records in different LV's, with different page sizes. The client controls the number of LV's, and which page goes into which LV. If the client prefers, all pages can be in one large LV. A transaction can read or write pages from any number of LV's, without penalty.

The IS assigns a permanent page number to each page. The combination of an LV number and a page number within that LV uniquely specifies a page. Page numbers are allocatable resources: the client specifies the LV, and the IS responds with a new page number. The client can then read and write this page. A page retains its page number over updates. As with page-writes, every page-allocate or page-free request must be for some active transaction, and they don't become permanent until that transaction commits.

Think of page numbers as a virtual address space which the IS provides on top of the physical address space of disk blocks. Page numbers are not physical disk block addresses, and locality of page number does not imply physical locality.

The client can embed page numbers in other pages. For example,

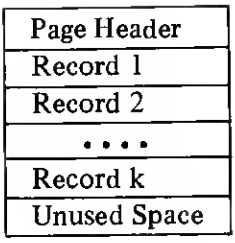

Fig. 6—Page format.

consider a database manager which uses a B-tree for the primary index.[24] It might use three LV's: one for the actual data records, another for the B-tree, and one more for secondary indexes. Pages in the B-tree LV would contain page numbers. Leaf pages would contain page numbers in the data LV, as would pages in the secondary index LV. Non-leaf pages in the B-tree LV would have page numbers for other pages in that LV.

4.3 Page format

The content-addressable version of the IS uses a page format based on that of the underlying search disk. As in Fig. 6, a page consists of a header, a variable number of records, and some unused space. The page header has the page number, LV number, cell number (see Section 4.4), number of records, last update time, etc. The client can put as many (or as few) records as desired in a page, up to the maximum size of the page.

A record consists of a header, followed by a variable number of self-defining attribute-value pairs, as in Fig. 7. The record header has the page number, and (perhaps) some other data. The record body is a set of triples (attribute-number,length,value). Values are inherently variable length, as are records. Not all attributes need be present in every record. In particular, triples with null values may be omitted. The attribute numbers are 16-bit integers; 8-bit identifiers are not sufficient. The rest of this paper will use attribute names instead of numbers. Assume the client provides a simple name-to-number map. The IS reserves a few attribute numbers for its own use.

While the IS allows retrieval based on the contents of records, update is by page. To update a record, the client reads the page containing that record, and then rewrites the entire page. The client can add, delete, grow, or shrink records at any time. However, if the client wants to grow a record and the page does not have enough unused space, the client must find a new page (or whatever) for the record.

For now, the conventional-disk version of the IS does not impose this page format. Instead, it just receives a fixed-size header (primarily

Record Header		
attr 1	len 1	value 1
attr 2	len 2	value 2
	....	
attr m	len m	value m

Fig. 7—Record format.

for sanity checking), and treats the rest of the page as an uninterpreted array of bytes.

4.4 Cells

When an LV is created, the IS reserves a contiguous set of disk blocks for it. These blocks are partitioned into a set of disjoint cells. The system administrator specifies the number of cells, and the number of blocks per cell. Typically, cells are on the scale of a disk cylinder (10 to 20 tracks). Different LV's can have different cell sizes, and different numbers of cells. Each cell is identified by a combination of LV number and cell number within that LV.

There are two reasons for the cell concept:

(*i*) Cells offer physical locality. That is, the client can access a set of pages faster if the pages are all in the same cell (assuming no other accesses intervene). This applies to both the content-addressable and conventional-disk versions of the IS.

(*ii*) For the content-addressable IS, cells are the minimal unit of content-addressability. The IS can search all pages in cell C in some short unit time (typically on the order of one revolution of a disk). Searching 10 cells takes 10 unit times.

Each page is in one (and only one) cell. The client controls this mapping:

(*i*) When allocating a new page, the client must specify a cell. If that cell is full, the IS refuses. The client can then try another cell, or ask the IS to put it "anywhere."

(*ii*) To read a page, the client needs only the page number. The cell number is not embedded in the page number; the IS keeps a separate map of page-to-cell numbers.

(*iii*) The IS never moves pages by itself. Once allocated in a cell, the IS will keep the page in that cell, regardless of how often the page is written or bow full the cell becomes.

(*iv*) The client can explicitly rearrange pages. As an off-line operation, the client can pick up all existing pages, and spread them out among the cells in a different pattern. The page numbers do not change.

Thus each page has a page number and a cell number. The cell number is transient: it's good during a transaction, but tomorrow it may be different. The page number is permanent, and is retained even if the client moves the page to a different cell.

There are two ways a database manager client can use cells. The first method takes advantage of the locality offered by cells, and tries to keep related pages in the same cell to improve performance. For example, suppose that the client uses one LV for indexes, and another for data records. The index LV has page numbers for the data LV. The index does not have cell numbers; it's valid no matter how pages are spread over cells. When allocating a new page, a strategy module in the client picks a cell with related pages. If the IS says that that cell is full, the client puts the page anywhere. After "too many" pages have been put "anywhere," the client reorganizes the data LV. Because data pages retain their page numbers, the client can reorganize the data LV without changing the index LV. Section 4.5 gives a more detailed example.

The other technique only works with the content-addressable version of the IS. The client defines formal clusters, maps those clusters to cells, and uses the content-addressble features to search clusters. For example, suppose that the NAME attribute is the primary key. The client might put all names that start with A or B in cell 1, C or D in cell 2, etc. The advantage is that the client doesn't need an index for the primary key. To find Baker, the client searches cell 1 for NAME = "Baker". The problem comes when cell 1 is full, and the client needs to add a record for Bell. Then the client must immediately find another cell for A-B records (an overflow cell, perhaps). Eventually, the client would reorganize, perhaps this time putting A records in cell 1, B records in cell 2, etc.

4.5 Example: A B-tree application

Suppose that a database manager (DBM) has a large collection of records, and uses a B-tree index for the primary key.[24] The DBM uses two LV's: one for the B-tree pointers, and another for the data records. Each page in the pointer LV contains just one record, with one attribute: a large array of keys and page numbers (pointers). A page in the data LV contains a variable number of records. Leaf pages in the pointer LV point to pages in the data LV, while non-leaf pages point to other pages in the pointer LV.

The pointer LV is much smaller than the data LV, perhaps only a few cells. The DBM doesn't need to worry about which cell to allocate a new pointer page from. In fact, the pointer LV might have only one cell.

Let's suppose that the DBM often does sequential processing on the

records, and would like records to be in key order in the data LV. In other words, the records in cell 1 should precede those in cell 2, which should precede those in cell 3, and so forth. Suppose that the data LV is partially filled, and pages are in the desired order. To maintain this ordering, the DBM does the following when inserting a new record:

(*i*) Using the pointer LV, the DBM finds the page in the data LV that contains the records with the closest primary key values.

(*ii*) If the new record will fit in that data page, the DBM updates it, and is done.

(*iii*) If not, the DBM allocates a new data page, and splits the records in the old page. The DBM first tries to allocate the new data page from the same cell as the old page. If that cell is full, the DBM uses any cell.

(*iv*) The DBM inserts a pointer to the new data page into the leaf page of the pointer LV. If it won't fit, the DBM splits pointer pages, according to the B-tree algorithm.

Thus, the data LV stays in order for a while, but as cells become full, it becomes scrambled. At some point, the DBM decides to reorganize the data LV. The DBM can use the B-tree to retrieve all pages in key order. The IS recreates the LV from this stream, starting a new cell every n pages. After the reorganization, the data LV is in the desired order, and the DBM starts over. The DBM does not have to change the pointer LV.

When reorganizing, the DBM must decide how many pages to put in each cell. This may depend on the maturity of the database. For example, suppose that eventually the database will have 1,000,000 records, but it starts from zero and is loaded at the rate of 50,000 records per week. If reorganizations are done weekly, for the first few weeks the DBM might only load cells half full when reorganizing (assuming the records arrive in random key order). But after most of the records have been loaded, the DBM would pack more records per cell.

Note that the DBM doesn't worry about cells. The DBM just asks the IS to put new pages in the same cell as existing pages. The IS keeps track of how full cells are; the DBM doesn't.

V. CONTENT-ADDRESSABLE IMPLEMENTATION

This section gives the highlights of how the content-addressable version of the IS is implemented.

Pages do not share physical disk blocks. The rest of this section assumes a variable-format disk whose block size has been set to the maximum page size. Given a fixed-format disk, the IS stores pages in adjacent blocks, and reads or writes them as a unit.

For each LV, the IS has a "page map," which is large array of cell numbers indexed by page numbers. The page map also controls page

number allocation. The map is only updated when a page number is allocated or freed; it is not updated every time an existing page is rewritten. The IS keeps the map on a conventional disk (or whatever), with a cache in main memory. The information in the map is duplicated in the header of each page. Thus, if part of the map is corrupted, the IS can recreate it by searching all cells in the LV.

This IS also has a simple map from cell numbers to physical cylinder addresses. This doesn't change while the IS is active and it is small enough to fit in main memory.

The IS puts the page number in a searchable field in each block. Thus, to read page P, the IS first gets P's cell number from the page map, seeks to the corresponding cylinder, and then orders the disk to retrieve the block whose PAGE field has the value P.

Note that the page map is a cell-level index for pages. The IS does not have an index from page numbers to specific disk blocks.

When given a write-page request for page P, the IS writes the new version in a free block in P's cell. Actually, the IS just schedules the disk write operation, and then tells the client task that the request is complete. The actual writing can be overlapped with the client's think-time for the next request. Note that the IS writes the new version before the transaction commits, without removing the old version. Therefore at any one time there may be several different versions of page P.

Before explaining how the IS eventually removes those old versions, we need to introduce some terminology. A transaction T becomes a "grandfather" when all transactions that were active when T committed have themselves committed (or aborted). In Fig. 2, transaction A becomes a grandfather as soon as it commits, and transaction B becomes a grandfather when C aborts. Thus, transaction T becomes a grandfather when no other transaction needs the old versions of the pages that T updated. After T becomes a grandfather, the IS "retires" it, by freeing all disk blocks that hold the old versions of pages written by T. Transactions are retired in the order in which they committed. Retirement is a background process, which the IS can do at its leisure.

Suppose that a transaction that started at time t asks the IS to read page P. The IS first asks the disk to retrieve all versions of page P. Of those versions, the IS software selects the most recent one of those written by transactions that committed before time t. Thus, given an arbitrary version of P, the IS needs to know the commit-time of the transaction that wrote it. Because the IS writes new versions before commit, the IS can't just put the commit-time in the page header. Instead we have to be a bit more clever.

The IS assigns a permanent Transaction Serial Number (TSN) to each update transaction. The IS assigns a TSN to transaction T when

T writes its first page. The IS puts T's TSN in a field in every page that T writes. These numbers are never reused. At 10,000,000 transactions per day, 48-bit serial numbers will last for 77,000 years, which should be sufficient for Bell System use.

So now the problem is to map TSN's to commit-times. The IS keeps a list of the TSN and commit-time for all commited but unretired transactions. This is called the *commit-list*. Committed transactions are added at one end, and grandfathers are removed from the other as they are retired. Thus, the commit-list has just those transactions that have committed during the lifetime of the oldest uncommitted transaction. This should be at most a few hundred transactions.

Given a TSN, the IS first looks for it in the commit-list. If found, the IS uses the corresponding commit-time. If not, that transaction must be retired. We don't need to know the exact commit-time for a retired transaction; its is sufficient to know that (*i*) it committed before the oldest transaction in the commit-list, and (*ii*) it committed before any active transaction started. Furthermore, the retirement process ensures that for any given page P, there is only one version of P written by a retired transaction.

So much for reading and writing pages; what about atomic commits and crash recovery? First, note that the IS never updates a data block in place. The IS either writes a new version into a free block, before a transaction commits, or else frees an old block after its data is no longer needed. A corrupt block is always free, and no vital data will be lost when a block gets corrupted if the IS crashes while writing it. Thus, the IS dosen't need conventional undo-redo logs.[3]

The IS keeps two stable lists of TSN's. The first is the commit-list, described above. The other is the *active-list*, which has the TSN's of all active, uncommitted update transactions. These lists must be safe from the effects of crashes. For each list, the IS keeps two copies on disk, and carefully updates both copies.[25]

When transaction T writes its first page, the IS must ensure that T's TSN has been safely added to the active-list before any data blocks are written for T. Similarly, before removing T from the commit-list, the retirement demon must ensure that all the old blocks have been successfully marked as free (i.e., all disk writes have completed).

To commit transaction T, the IS first waits for any outstanding disk writes for T to complete, and then safely moves T's TSN from the active-list to the commit-list. Adding the TSN to the commit-list is what really commits a transaction. Note that there is no burst of I/O at commit. The disk writes are overlapped with the client's think-time, so they are usually complete when the client issues the commit request.

To abort a transaction, the IS just frees all disk blocks it wrote, and then removes its TSN from the active-list.

To recover from a crash, the IS first reads the active-list and commit-list from disk. The IS then aborts all transactions in the active-list, and retires all transactions on the commit-list. These operations can be overlapped with normal processing.

That, in a nutshell, is how the IS works. Note several advantages:

- There is no burst of I/O at commit.
- Disk blocks holding pages are never updated in place.
- Each version of a page is written twice in its lifetime: once when it's initially written, and once to mark it as free, when it's retired. Both of these writes can be overlapped with other operations.

For this scheme to work efficiently, the IS needs a pool of free blocks in each cell. The size of the pool depends on how many pages each transaction updates, and on the running-time of transactions. To help maintain a free pool, when an LV is created, the system administrator specifies a limit on the number of distinct pages that can be allocated in each cell. For example, if cells have 1000 disk blocks, the administrator may limit each cell to 950 distinct pages.

If a cell does run out of free blocks, the IS automatically uses a shared overflow area. This is a transient condition, and it's invisible to the client. Eventually, the IS can always move the overflow blocks back to their desired cells. However, excessive use of the overflow area will degrade performance.

The administrator specifies the size of the overflow area when configuring the IS. This size, plus the amount of free space in cells, limits the total number of pages that can be written during the life of the longest-running transaction (see Section 2.1). If this limit is exceeded, the IS can always abort the oldest transaction. This makes some transaction a grandfather, and retiring it should free some disk blocks.

VI. CONVENTIONAL DISK IMPLEMENTATION

6.1 Overview

The conventional-disk version of the IS also uses a shadowing scheme. A simple page index tells which disk block has the current version of each page. These disk blocks are not updated in place. Instead, when transaction T writes a page, the IS writes the new version to a free disk block, and saves the page and block numbers in a list for transaction T. When T commits, the IS uses that list to carefully update the page index. For each page which T updates, the IS saves the block number with the old (replaced) version in another list, for possible use by any transactions which started before T committed. These old blocks will not be reused until the last such transaction has terminated.

6.2 Disks and disk drivers

For now, the conventional disks are fixed-sector disks, with 512-byte blocks. A disk driver task provided by the network kernel does all disk I/O. To read or write a block, the IS sends a message to the disk driver, specifying the command, the disk address, the main memory address, and the number of blocks to read or write. When done, the disk driver sends a reply message to the IS. The IS can have multiple read/write requests active simultaneously. Note the division of responsibility: the IS handles caching, and the disk driver handles scheduling.

6.3 Page maps and page frames

Each LV has two disk structures: a page map and a set of page frames. The page frames occupy a set of contiguous disk blocks. The frames are arbitrarily partitioned into physical cells, each with the same number of frames. A page frame is addressed by an encoding of its physical cell number and frame number within that cell. The size of a page frame, and hence the size of a page, is limited to a multiple of 512 bytes.* A page frame is a contiguous set of disk blocks, which the IS reads or writes as a unit.

The page map is a single-level index of pages, with one 32-bit entry per page number. A free page has a null entry; an allocated page has an encoding of the page's logical cell number, and a physical cell/frame number. This is the frame with the current version of the page. The IS reads and writes the page map in 512-byte blocks (at least for now), using a main-memory cache. This cache is separate from that used for page frames.

At any one time, the page map defines the current, consistent state of the database. The IS updates the map very carefully, and only when committing a transaction (see below).

For each LV, the system administrator specifies the maximum number of pages, the page size, number of cells, and number of page frames per cell. The administrator also specifies where the page map and page frames are on disk. They can be on separate disks. For example, the administrator might put the page map on a faster disk, or even on a fixed-head disk. The space for the page map must be contiguous. For now, the set of page frames must also be contiguous and on one disk, but this could be changed.

6.4 Page frame allocation

Allocation of page frames is controlled by a bit-map. This bit-map

* Actually, the page size is slightly less than a multiple of 512, because the IS steals a header from each page frame. The current header is 12 bytes, and contains a time stamp, the logical cell/page number, and the physical cell/frame number. The header is just for sanity checking; it's not needed for correct operation or for crash recovery.

reflects the shadow state as well as the current consistent state. The "consistent state" contains all frames in the current page map. The "shadow state" includes the new frames written by uncommitted transactions, and the old frames replaced by committed transactions.

This bit-map is not safe over crashes. Currently it's kept only in main-memory. The IS could swap the bit-map, but that doesn't seem worthwhile. For example, consider an IS with 10**6 frames. If frames are 2K bytes, that's 2 gigabytes of disk space. The corresponding bit-map only takes 128K bytes. Since main-memory costs are dropping relative to disk costs, and since fast access to the bit-map is important for performance, swapping the bit-map doesn't seem cost-effective.

The bit-map is recreated at boot-time. The IS first clears the bit-map, and then reads the entire page map. For each page frame in the map, the IS sets its allocation bit. Just for sanity, the IS also verifies that the bit was not set (this should never happen, but it's a cheap test for duplicate entries, so why not?). Since the page map is contiguous, the IS can read it very efficiently; I estimate that the IS could read a page map with 10**6 entries in less than one minute. However, the I/O time is dominated by the CPU time to set the bits. For a full 10**6 page database, the current version of the scan code takes about 8.5 minutes on a 3B-20. It's not great, but it's not too bad. Most of the time is spent in a few, small C functions. If desired, those functions could be written in assembler, or even in microcode.

Currently, the IS does this scan at boot-time, and locks out all requests until it's complete. This simplifies the module which allocates page frames. However, if the initial scan delay isn't acceptable, that module could be modified for parallel scanning; it would automatically delay any allocation request (i.e., a page write) until that LV's page map had been scanned. Thus the IS could allow read requests immediately.

6.5 Lists

For each active transaction, the IS keeps a list of its important pages (the "important-list"), and a list of the pages it wrote (the "write-list"). These lists are controlled by a generalized list manager module within the IS. The list manager provides append and scan access, and automatically frees lists for terminated transactions. The list manager tries to keep these lists in main memory, transparently swapping out the least recently used lists as necessary. The important points are (*i*) these lists are not safe over crashes, and (*ii*) the IS can handle very large lists.

6.6 Writes, commits, and recovery

The IS does the following to write page P for transaction T:

1. Allocates a new page frame F.
2. Gets a buffer for F, and copies the client's data into it.
3. Adds the pair (P, F) to T's write-list.
4. Sends the reply message to the client ("It's done, boss!").
5. Does the actual disk write. Note that this overlaps the client's subsequent processing.

The IS does the following when committing transaction T:

1. Waits for all page frame writes for T to complete.
2. Verifies that no transaction that committed after T started wrote one of T's important pages (more on this later). If this tests fails, the IS aborts T by freeing all page frames in T's write-list.
3. Scans T's write-list, and for each pair (P, F):
   a. Reads the page map block for page P into the cache, if it isn't there already.
   b. Sets Fx to the old frame number in P's page map entry.
   c. Updates P's page map entry in the cache, but does not update the disk copy.
   d. Adds the pair (P, Fx) to a "prior-value" list being created for T.
4. Writes the new versions of all updated page map blocks to a reserved set of disk blocks, known as the "Intention-List".[3]
5. Updates the actual page map blocks.
6. Clears the Intention-List.
7. Adds T's prior-value list to the set of prior-value lists for all committed but unretired transactions (see below).

At boot-time, the IS just reads the Intention-List. If it indicates that a commit was in progress, the IS updates the indicated page map blocks. Note the following points:

(*i*) The Intention-List ensures that either all the T's pages will be updated or else none will.

(*ii*) If a failure occurs when writing a page map block, the entire block might be destroyed. This is why the Intention-List has full blocks, not just T's write-list.

(*iii*) The Intention-List has a fixed maximum size (say 20 page map blocks). This is fine for most transactions, but some transactions will need to update more page map blocks. Therefore if the IS fills the Intention-List before exhausting T's write-list, the IS first saves T's write-list in a reserved place on disk. The IS then writes the Intention-List, updates the page map blocks, and resets the main-memory copy of the Intention-List. The IS then continues scanning T's write-list, forming more Intention-Lists as needed. When done, the IS resets the disk copy of the write-list. During recovery, the IS first reads the Intention-List, updates the indicated page map blocks (if any), and resets the Intention-List. Then if there is a saved write-list, the IS

reads it and updates the page maps accordingly, writing updated page map blocks to the Intention-List as before. When done, the IS resets the saved write-list. Note that another crash during recovery won't hurt anything.

(*iv*) For now, the Intention-List lives on a conventional disk, at a location specified by the administrator. Ideally, the Intention-List would live in a very fast nonvolatile memory— perhaps a RAM with battery backup.

(*v*) The page frames with the old values aren't freed immediately; they won't be until all transactions that started before T committed have terminated (see below).

(*vi*) Steps 1 and part of 2 can be done in parallel, but the rest must be done for one transaction at a time.

6.7 Grandfathers, retirement, and prior-value lists

A transaction T becomes a "grandfather" when all transactions that were active when T committed have themselves committed (or aborted). In Fig. 2, transaction A becomes a grandfather as soon as it commits, and transaction B becomes a grandfather when C aborts. Thus, transaction T becomes a grandfather when no other transaction needs the old versions of the pages which T updated. The IS keeps the prior-value lists for all committed transactions which have not yet become grandfathers.

When T becomes a grandfather, the IS "retires" it, by freeing all page frames on T's prior-value list, and then discarding T's list. Note that retirement is a background process, which the IS can do at its leisure.

6.8 Read requests

To read page P for transaction T, the IS first determines which page frame F has the correct version of P. The IS uses the following algorithm:

1. Searches T's write-list for P.
2. Otherwise, searches the prior-value lists (see above) for all transactions that have committed after T started. If more than one has an entry for P, uses the one from the earliest transaction.
3. Otherwise, uses the page map entry for P.

The IS then reads frame F into a buffer and returns it to the client. The frame buffers are cached, of course.

For now, list searching is done sequentially. Thus, the search time depends on the number of pages written by T, and on the number of transactions that have committed during T's lifetime. Large transactions may have a high overhead, but small, fast transactions have little overhead.

The IS uses the following technique to avoid searching the write-list unnecessarily. Let $h(P)$ be a hashing function from page numbers to the integers 1:N. For each transaction T, the IS keeps an N-bit bit-map. The bit-map is zeroed when T starts, and discarded when T terminates. When T writes page P, the IS sets bit $h(P)$; when T reads P, the IS searches T's write-list iff bit $h(P)$ is set.

To avoid searching the prior-value lists unnecessarily, the IS keeps a set of N counters. Counter $h(P)$ is the number of entries (in all prior-value lists) whose page number hashes into $h(P)$. Thus, the IS searches for page P iff the counter $h(P)$ is non-zero. When committing a transaction, the IS increments the counters for all pages on its list; when retiring a transaction, the IS decrements those counters. Counters are small (currently 4 bits). A counter is "frozen" when it hits the maximum value, and further increments or decrements have no effect. When too many counters become frozen, the IS resets all counters, and recreates them by scanning all prior-value lists.

The write-lists and prior-value lists are managed by modules that provide add/lookup access. These modules hide the bit-map and counter tests. Other techniques could be added without changing the rest of the IS.

6.9 Checking for interference

Before committing transaction T, the IS first verifies that no other transaction which committed after T started has written one of T's important pages. The IS does this by intersecting T's important-page list with the prior-value lists of all transactions which committed after T started. The IS currently uses the following algorithm:

1. If no transactions have committed since T started, we're done; T can be committed.
2. For each page P in T's important-page list, adds P to a work array iff the prior-value hash counter $h(P)$ is non-zero (see above).
3. If the work array is empty, we're done; T can be committed. Otherwise, sorts the work array by page number.
4. For each page P in the prior-value list of any transaction that committed after T started, tests if P is in the work array (uses binary search). Stops if any P is in the work array; T cannot be committed.

VII. ACKNOWLEDGMENTS

The Intelligent Store is the result of several discussions between myself and many others: initially, Bill Burnette, Dan Fishman, and John Linderman, and later Jean Benisch, Rudd Canaday, Alan Feuer, Joe Haggerty, Dave Nowitz, and Charles Wetherell. At this point, it's impossible to remember who first suggested what, but I do remember that Rudd suggested the concept of the IS as something common to a

file manager and a database manager, and John and Bill Burnette suggested parts of the consistency model.

REFERENCES

1. W. A. Burnette, R. H. Canaday, and D. H. Fishman, unpublished work, March 22, 1982.
2. K. P. Eswarin et al., "The Notions of Consistency and Predicate Locks in a Database System," Commun. ACM, *19*, No. 11 (November 1976), pp. 624–33.
3. J. N. Gray, "Notes on Data Base Operating Systems," in Operating Systems: An Advanced Course, R. Bayer et al. (ed), New York: Springer-Verlag, 1978, pp. 393–481.
4. H. T. Kung and J. T. Robinson, "On Optimistic Methods for Concurrency Control," ACM Trans. Database Systems, *6:2* (June 1981), pp. 213–26.
5. J. Banerjee, D. K. Hsiao, and R. J. Baum, "Concepts and Capabilities of a Database Computer," ACM Trans. Database Systems, *3:4* (December 1978), pp. 347–84.
6. J. Banerjee, D. K. Hsiao, and K. Kannan, "DBC—A Database Computer for Very Large Databases," IEEE Trans. Computers, *C28:6* (June 1979), pp. 414–29.
7. K. Kannan, "The Design of a Mass Memory for a Database Computer," Proc. Fifth Annual Symp. on Computer Architecture, April 1978, pp. 44–51.
8. V. A. J. Maller, "The Content Addressable File Store—CAFS," ICL Technical J, November 1979, pp. 265–79.
9. F. Baskett, J. H. Howard, and J. T. Montague, "Task Communication in DEMOS," Proc. Sixth Symp. on Operating Systems Principles, November 1977, pp. 23–31.
10. M. H. Soloman and R. A. Finkel, "The Roscoe Distributed Operating System," Proc. Seventh Symp. on Operating Systems Principles, December 1979, pp. 108–14.
11. D. J. DeWitt, "DIRECT—A Multiprocessor Organization for Supporting Relational Database Management Systems," IEEE Trans. Computers, *C28:6* (June 1979), pp. 395–406.
12. S. A. Schuster, H. B. Nguyen, and E. A. Ozkarahan, "RAP.2—An Associative Processor for Databases and Its Applications," IEEE Trans. Computers, *C28:6* (June 1979), pp. 446–58.
13. D. E. Shaw, "A Relational Database Machine Architecture," ACM Fifth Annual Workshop on Computer Architecture for Non-Numeric Processing," March 1980, pp. 84–95.
14. D. C. P. Smith and J. M. Smith, "Relational Database Machines," IEEE Computer, *12:3* (March 1979), pp. 28–38.
15. S. Y. W. Su, "Cellular Logic Devices: Concepts and Applications," IEEE Computer, *12:3* (March 1979), pp. 11–25.
16. S. Y. W. Su et al., "Architectural Features and Implementation Techniques of the Multicell CASSM," IEEE Trans. Computers, *C28:6* (June 1979), pp. 430–45.
17. D. J. DeWitt and P. B. Hawthorn, "A Performance Evaluation of Database Machine Architectures," Seventh Int. Conf. On Very Large Databases, September 1981, pp. 199–214.
18. G. G. Langdon, "A Note on the Associative Processors for Data Management," ACM Trans. Database Systems, *3:2* (June 1978), pp. 148–58.
19. D. S. Kerr, "Data Base Machines for Large Content-Addressable Blocks and Structural Information Processors," IEEE Computer, *12:3* (March 1979), pp. 64–79.
20. J. S. M. Verhofstad, "Recovery Techniques For Database Systems," ACM Computing Surveys, *10:2* (June 1978), pp. 167–95.
21. D. R. Ries and M. Stonebraker, "Effects of Locking Granularity in a Database Management System," ACM Trans. Database Systems, *2:3* (September 1977), pp. 233–46.
22. D. R. Ries and M. Stonebraker, "Locking Granularity Revisited," ACM Trans. Database Systems, *4:2* (June 1979), pp. 210–27.
23. H. Sturgis, J. Mitchell, and J. Israel, "Issues in the Design and Use of a Distributed File System," ACM Operating Systems Rev., *14:3* (July 1980), pp. 55–69.
24. D. Commer, "The Ubiquitous B-Tree," ACM Computing Surveys, *11:2* (June 1979), pp. 121–37.
25. B. W. Lampson and H. E. Sturgis, "Crash Recovery in a Distributed Data Storage System," Xerox-PARC Technical Report, April 27, 1979.

APPENDIX

Let X be a set with m distinct members, x_i. Assume that the x_i have been uniformly and independently selected from a universe of N elements. Let Y be a set with n distinct members, y_i, which have been selected from the same universe, independently of each other and of the x_i. We want to show that:

$$P(X \cap Y \doteq \phi) = \prod_{i=0}^{n-1} \left(1 - \frac{m}{N-i}\right) = \prod_{i=0}^{n-1} \frac{N-(m+i)}{N-i}.$$

The probability that X does not intersect Y is defined as:

$$P(X \cap Y = \phi) = P(y_1 \notin X \wedge y_2 \notin X \wedge \cdots \wedge y_n \notin X).$$

We will use induction on the number of terms in the right-hand side. To simplify notation, let's define A_k as:

$$A_k \equiv y_1 \notin X \wedge y_2 \notin X \wedge \cdots \wedge y_k \notin X.$$

Then the initial step is:

$$P(y_1 \notin X) = P(A_1) = 1 - P(y_i \in X) = 1 - \frac{m}{N}.$$

This follows from the fact that X has m uniformly and independently selected elements. For the induction step, we first assume that:

$$P(A_k) = \prod_{i=0}^{k-1} \left(1 - \frac{m}{N-i}\right)$$

and then show that relation holds for A_{k+1}. Using the definition of conditional probability, we get:

$$P(A_{k+1}) = P(y_{k+1} \notin X | A_k) P(A_k).$$

The A_k condition tells us that none of the first k elements are in X. That is, the m elements of X have really been selected from a universe of $N-k$ elements. Therefore, the conditional probability that y_i isn't in X is:

$$P(y_{k+1} \notin X | A_k) = 1 - \frac{m}{N-k}.$$

Substituting back and using the induction assumption gives:

$$P(A_{k+1}) = \left(1 - \frac{m}{N-k}\right) \prod_{i=0}^{k-1} \left(1 - \frac{m}{N-i}\right) = \prod_{i=0}^{k} \left(1 - \frac{m}{N-i}\right).$$

This proves the induction step. We can simplify things by observing that:

$$1 - \frac{m}{N-i} = \frac{N-(m+i)}{N-i}.$$

This gives as the final result:

$$P(X \cap Y = \phi) = \prod_{i=0}^{n-1} \frac{N-(m+i)}{N-i}.$$